ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Решение задачи о рюкзаке»

Выполнил работу

Маркелова Софья

Академическая группа №С3100

Принято

Ментор, Вершинин Владислав

Санкт-Петербург

2024

Введение:

Цель работы – решить задачу о рюкзаке тремя способами: полным перебором, методом жадного алгоритма и методом динамического программирования. Условия задачи звучат следующим образом, существует рюкзак, который может выдержать определённый вес (максимальная грузоподъемность). Также есть набор предметов, каждый из которых имеет свою стоимость и вес. Задача заключается в том, чтобы определить, какие предметы следует положить в рюкзак, чтобы максимизировать общую стоимость, не превышая при этом максимальную грузоподъемность рюкзака.

Для достижения цели были установлены следующие задачи: реализовать решение полным перебором с помощью рекурсии, решение методом жадного алгоритма нужно реализовать, воспользовавшись сортировкой и циклом, а в методе динамического программирования стоит использовать таблицу.

1. Теоретическая подготовка:

Для выполнения работы необходимо обладать теоретической подготовкой в части понимания работы с векторами, рекурсией. Также потребуется понимание в работе с таблицами для решения задачи методом динамического программирования.

1. Реализация:

В процессе реализации были использованы стандартная библиотека iostream для вывода результата, algorithm для работы со стандартным набором алгоритмов. В качестве входного типа данных был выбран vector. Также использована структура Item для хранения информации о каждом предмете, который потенциально можно положить в рюкзак. Она содержит два поля: value - стоимость предмета и weight – его вес.

struct Item {

    int value;  // ценность

    int weight; // вес

};

Решение задачи методом полного перебора

Для решения задачи была реализована функция knapsack, которая принимает 3 параметра: capacity – максимальная грузоподъёмность рюкзака, items – вектор предметов, доступных для выбора, n – количество предметов, доступных для рассмотрения.

int knapsack(int capacity, const vector<Item>& items, int n)

Вначале рассматривается бызовый случай: если нет предметов (n == 0) или если рюкзак не может вместить ни одного предмета (capacity == 0), то максимальная ценность, которую можно получить, равна 0. Мы возвращаем 0, так как ничего нельзя взять.

if (n == 0 || capacity == 0) {

        return 0;

    }

Следующим шагом происходит проверка веса текущего предмета: превышает ли вес текущего предмета (items[n – 1]) грузоподъемность рюкзака. Если да, то данный предмет нельзя положить в рюкзак, следовательно, вызываем функцию рекурсивно, уменьшая количество рассматриваемых предметов на 1 (n - 1). Таким образом, мы просто пропускаем этот предмет и продолжаем искать решение среди оставшихся.

if (items[n - 1].weight > capacity) {

        return knapsack(capacity, items, n - 1);

Если же вес текущего предмета меньше или равен грузоподъемности рюкзака, то рассматривается два варианта. Первый – это включение текущего элемента. Мы добавляем стоимость текущего предмета (items[n - 1].value) к результату рекурсивного вызова функции с обновленной грузоподъемностью (capacity - items[n - 1].weight) и уменьшаем количество оставшихся предметов на 1 (n - 1). Второй – исключение текущего элемента. Мы просто вызываем функцию с той же грузоподъемностью (capacity) и уменьшаем количество оставшихся предметов на 1 (n - 1). После чего мы используем max для выбора наибольшей ценности из двух вариантов — включения или исключения текущего предмета.

else {

        // Возвращаем максимум между включением и исключением текущего предмета

        return max(

            items[n - 1].value + knapsack(capacity - items[n - 1].weight, items, n - 1), // Включаем предмет

            knapsack(capacity, items, n - 1) // Исключаем предмет

        );

    }

Решение задачи методом жадного алгоритма

Изначально была реализована булевая функция compare для сравнения двух предметов на основе их соотношения ценности к весу. Она принимает два параметра: Item a - первый предмет, Item b - второй предмет. Логика функции следующая: функция возвращает true, если соотношение ценности к весу первого предмета (a.value / a.weight) больше, чем у второго (b.value / b.weight). Это означает, что предмет a более "выгоден" по сравнению с предметом b в контексте жадного алгоритма. Данная функция будет передана в стандартную функцию sort, чтобы отсортировать вектор предметов по убыванию их соотношения ценности к весу.

bool compare(Item a, Item b) {

    return (double)a.value / a.weight > (double)b.value / b.weight;

}

Функция knapsack\_greedy реализует жадный алгоритм для решения задачи о рюкзаке. Она пытается максимизировать общую ценность предметов, которые можно положить в рюкзак, используя жадный подход. Она прнимает два параметра: capacity - максимальная грузоподъемность рюкзака, items - вектор предметов, доступных для выбора.

int knapsack\_greedy(int capacity, vector<Item>& items)

Следующим шагом происходит сортировка вектора items по убыванию соотношения ценности к весу с помощью функции compare. Это позволит сначала рассмотреть наиболее "выгодные" предметы (с наибольшей ценностью на единицу веса).

sort(items.begin(), items.end(), compare);

Затем происходит инициализация переменной для хранения общей ценности предметов, которых можно положить в рюкзак.

int totalValue = 0;

Основной цикл проходит по каждому предмету в отсортированном векторе items. Если оставшаяся грузоподъемность рюкзака (capacity) меньше или равна нулю, это означает, что рюкзак заполнен, и мы выходим из цикла.

for (const auto& item : items) {

        if (capacity <= 0) break;

Происходит проверка возможности добавления предмета. Если текущий предмет помещается, мы добавляем его ценность к totalValue и уменьшаем оставшуюся грузоподъемность рюкзака на вес этого предмета (capacity -= item.weight). В противном случае, мы просто пропускаем его и переходим к следующему предмету.

if (item.weight <= capacity) {

            // Если текущий предмет помещается в рюкзак, добавляем его целиком

            totalValue += item.value;

            capacity -= item.weight; // Уменьшаем оставшуюся грузоподъемность

        }

В конце возвращается итоговая сумма, которую удалось получить.

 return totalValue;

Решение задачи методом динамического программирования

Для решения задачи была реализована функция knapsack\_dp, которая находит максимальную ценность предметов, которые можно положить в рюкзак с заданной грузоподъемностью. Она принимае два параметра - capacity: максимальная грузоподъемность рюкзака, items - вектор предметов (с их ценностями и весами), доступных для выбора.

int knapsack\_dp(int capacity, const vector<Item>& items)

Инициализируется переменная n – количество предметов. Её будем использовать в циклах и при создании таблицы динамического программирования.

int n = items.size();

В следующем этапе создаётся таблица динамического программирования – двумерный вектор dp, который будет хранить максимальные ценности для различных подзадач. Первая размерность (от 0 до n) соответствует количеству предметов (включая случай, когда предметов нет). Вторая размерность (от 0 до capacity) соответствует различным возможным грузоподъемностям рюкзака. Все элементы инициализируются значением 0, что означает, что если рюкзак пуст или нет предметов, максимальная ценность равна 0.

vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

Затем происходит заполнение таблицы циклом: внешний цикл (i) проходит по всем предметам от 1 до n, внутренний цикл (j) проходит по всем возможным грузоподъемностям рюкзака от 0 до capacity.

for (int i = 1; i <= n; ++i) {

        for (int j = 0; j <= capacity; ++j)

Происходит проверка возможности добавления предмета. Если текущий предмет (индекс i – 1) не помещается в рюкзак с грузоподъемностью j (items[i - 1].weight > j), то максимальная ценность для этой подзадачи будет равна максимальной ценности без этого предмета, то есть dp[i - 1][j]. Мы просто копируем значение из предыдущей строки.

if (items[i - 1].weight > j) {

                dp[i][j] = dp[i - 1][j];

Если текущий предмет помещается в рюкзак, мы должны выбрать между двумя вариантами: не брать текущий предмет: максимальная ценность будет равна dp[i - 1][j] или взять текущий предмет: максимальная ценность будет равна ценности этого предмета плюс максимальная ценность для оставшейся грузоподъемности (j - items[i - 1].weight), то есть dp[i - 1][j - items[i - 1].weight] + items[i - 1].value. Затем с помощью функции max мы выбираем максимальное значение из двух вариантов и сохраняем его в dp[i][j].

else {

                // Выбираем максимальную ценность: не брать предмет или взять его

                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - items[i - 1].weight] + items[i - 1].value);

            }

После того как мы заполнили всю таблицу, максимальная ценность, которую можно получить с помощью всех предметов при данной грузоподъемности, будет находиться в правом нижнем углу таблицы (dp[n][capacity]). Мы возвращаем это значение как результат функции.

return dp[n][capacity];

Заключение:

В ходе выполнения работы мною были реализованы три способа решения задачи о рюкзаке. При тестировании функций на одинаковых входных данных результаты совпали. Однако стоит отметить, что сложность подходов к решению отличается. Так, в методе полного перебора сложность составляет O(2^n) – экспоненциальный рост, что делает неэффективным решение при больших наборах данных. Сложность решения задачи методом жадношо алгоритма составляет O(n log(n)), сложность решения методом динамического программирования - O(n × capacity).

ПРИЛОЖЕНИЕ А

Листинг кода файла lab\_8.cpp

// Задача о рюкзаке

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Структура для хранения информации о предмете

struct Item {

    int value;  // ценность

    int weight; // вес

};

// 1. Полный перебор

// Функция для решения задачи о рюкзаке с использованием рекурсии

int knapsack(int capacity, const vector<Item>& items, int n) {

    // Базовый случай: если нет предметов или емкость рюкзака равна нулю

    if (n == 0 || capacity == 0) {

        return 0;

    }

    // Если вес текущего предмета больше емкости рюкзака, пропускаем его

    if (items[n - 1].weight > capacity) {

        return knapsack(capacity, items, n - 1);

    } else {

        // Возвращаем максимум между включением и исключением текущего предмета

        return max(

            items[n - 1].value + knapsack(capacity - items[n - 1].weight, items, n - 1), // Включаем предмет

            knapsack(capacity, items, n - 1) // Исключаем предмет

        );

    }

}

// 2. Жадный алгоритм

// Функция для сравнения предметов по соотношению ценности к весу

bool compare(Item a, Item b) {

    return (double)a.value / a.weight > (double)b.value / b.weight;

}

// Функция для решения задачи о рюкзаке методом жадного алгоритма

int knapsack\_greedy(int capacity, vector<Item>& items) {

    // Сортируем предметы по соотношению ценности к весу

    sort(items.begin(), items.end(), compare);

    int totalValue = 0; // Общая ценность, которую мы можем получить

    for (const auto& item : items) {

        if (capacity <= 0) break; // Если рюкзак заполнен, выходим из цикла

        if (item.weight <= capacity) {

            // Если текущий предмет помещается в рюкзак, добавляем его целиком

            totalValue += item.value;

            capacity -= item.weight; // Уменьшаем оставшуюся грузоподъемность

        }

        // Если предмет не помещается, просто пропускаем его

    }

    return totalValue; // Возвращаем общую ценность

}

// 3. Динамическое программирование

// Функция для решения задачи о рюкзаке методом динамического программирования

int knapsack\_dp(int capacity, const vector<Item>& items) {

    int n = items.size();

    // Создаем двумерный вектор для хранения максимальных ценностей

    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    // Заполняем таблицу динамического программирования

    for (int i = 1; i <= n; ++i) {

        for (int j = 0; j <= capacity; ++j) {

            // Если текущий предмет не помещается в рюкзак

            if (items[i - 1].weight > j) {

                dp[i][j] = dp[i - 1][j];

            } else {

                // Выбираем максимальную ценность: не брать предмет или взять его

                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - items[i - 1].weight] + items[i - 1].value);

            }

        }

    }

    return dp[n][capacity]; // Возвращаем максимальную ценность

}

int main() {

    // Инициализация данных

    int n = 3; // Количество предметов

    int capacity = 4; // Вместимость рюкзака

    vector<Item> items = {

        {3000, 4},   // Предмет 1: ценность 3000, вес 4

        {2000, 3},  // Предмет 2: ценность 2000, вес 3

        {1500, 1},  // Предмет 3: ценность 150, вес 1

    };

    // Вызов функции полного перебора

    int max\_value\_recursive = knapsack(capacity, items, n);

    cout << "Maximum value by recursion: " << max\_value\_recursive << endl;

    // Вызов жадного алгоритма

    int max\_value\_greedy = knapsack\_greedy(capacity, items);

    cout << "Maximum value by greedy algorithm: " << max\_value\_greedy << endl;

    // Вызов динамического программирования

    int max\_value\_dp = knapsack\_dp(capacity, items);

    cout << "Maximum value by dynamic programming: " << max\_value\_dp << endl;

    return 0;

}